GTU Department of Computer Engineering
CSE 222/505 - Spring 2022
HOMEWORK 2

Yusuf Arslan
200104004112

1. System Requirement

First of all the operating system should have Java virtual machine (JVM), Java development kit (JDK) and ‘at least’ Java runtime environment – 11 (JRE11).

The application can basically be defined as a street simulator. The user defines a street length and adds buildings on both sides of that street. He/She can then modify this street. Add or remove buildings, and ultimately print the street's silhouette.

The height and length of the building, the length of the street are restricted due to the constraints of the terminal screen. Details are available in the program documentation.

The user is given the opportunity to do keep buildings in different Java data collections and also the costum LDLinkedList, which is a linked list and I improved it as the part of the program.

2. Class Diagrams

Class diagrams attached to the and of the file.

3. Problem Solutions Approach

What is the problem?

We are asked to design and and implement a city planning software that will be used for designing a small street town. Street consists of two rows, equal length, and every row keeps constructions. This constructions should be kept in different data structures.

Beside the create the Building’s and add them into the street, we are expected to compare time complexity of the data structures used in the program.

Details:

There will be 4 type construction in the street, they are: House, office, market and playground. Every constructions has some properties beside the length and height.

House, represent with, room number, color and owner, Office, job-type (business) and owner,

Market, opening and closing time and owner.

Playground is a simple construction and it has default height.

So, in program there will be 2 modes,
1- Editing mode, that allows to modify street. (add and remove buildings to the rows)
2- Viewing mode, that allows to display street and itsproperties

What is my solution approach?

First of all, I brought House, Office, Market and Playground together under the roof of 'Building' class. I created a 'is a' relationship, and I extended classes for every construction from 'Building'.

They are, House, Office, Market and Playground classes.
All this sub-classes, besides the represent the super class <Building> they also has specific properties inside them.

Then,
I created one other class, to keep rows and buildings on the rows: Street
Street class basically is also a container class that has a length and two container that keeps reference of Building's which they are correspond to the rows. (row1, row2)

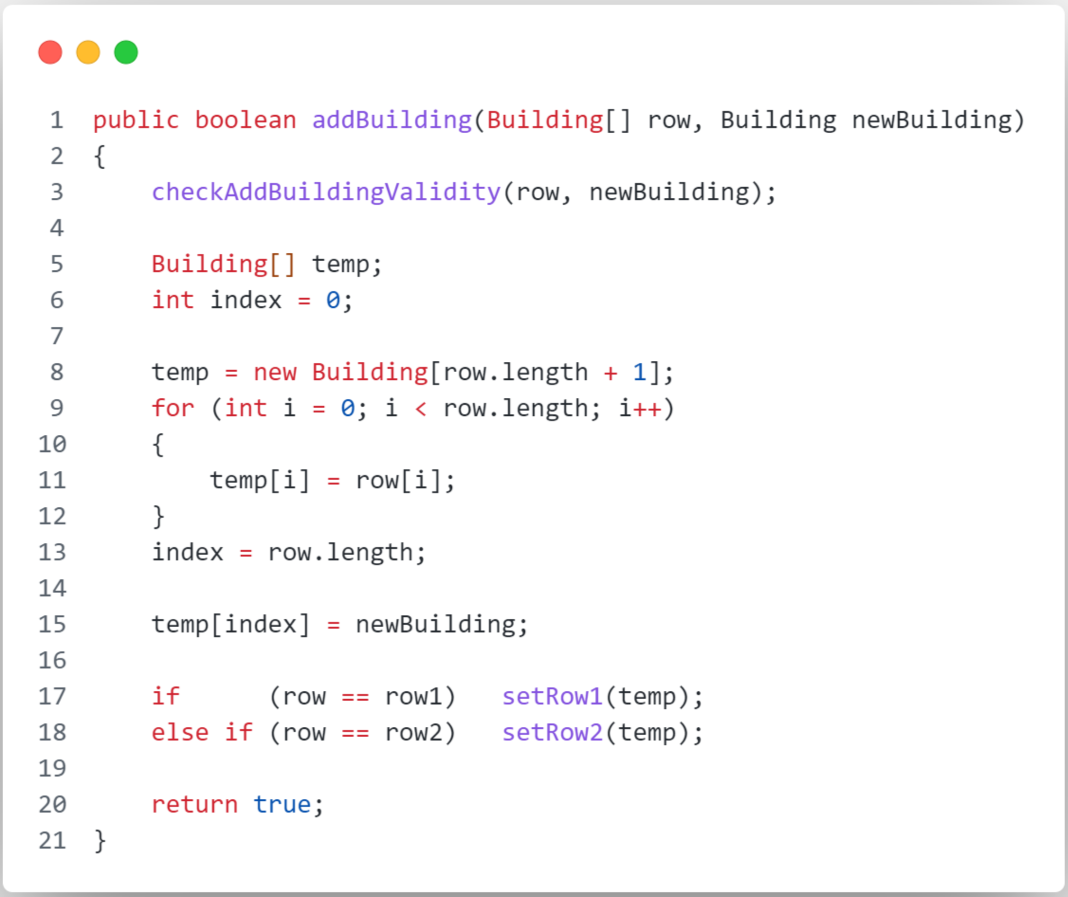
Finally,
The user, will be able to create Buildings, add position where it will be added to the street, and then add the building to the street if it is possible.

Let's go into the details a little more.

Below, we will examine the basic methods in Street classes that hold buildings in different data structures and then, calculate the time complexity of those methods.
We will later calculate the time complexity of the costum LDLinkedList class and its iterators.

- Impelementations of addBuilding() methods.

  Add a building to the street class with the basic array.



```java
public boolean addBuilding(Building[] row, Building newBuilding)
{
    checkAddBuildingValidity(row, newBuilding);

    Building[] temp;
    int index = 0;

    temp = new Building[row.length + 1];
    for (int i = 0; i < row.length; i++)
    {
        temp[i] = row[i];
    }
    index = row.length;

    temp[index] = newBuilding;

    if      (row == row1)   setRow1(temp);
    else if (row == row2)   setRow2(temp);

    return true;
}
```

To add a Building into the Street, we should check first if Street is available for the building.

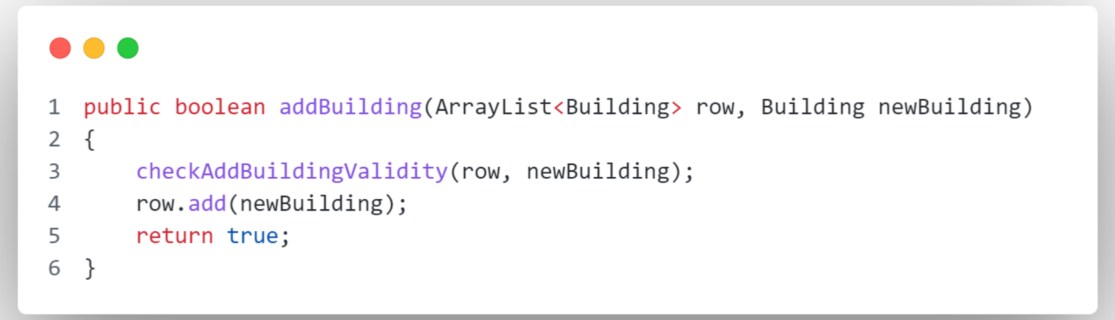
method checkAddBuildingValidity(), checks street and building length *θ(1)*
and if lengths are available, there is a loop that iterates one

time to control if the position that wants to inserted is occupied. *θ(n)*
If, all conditions satisfies array is resized, elements are shifted and the new item is inserted to that array. *θ(n)*

So, the total complexity of addBuilding() method for basic array will be: O(n)

Meanwhile, time of checkAddBuildingValidity() will be O(n) for all containers.

Add a building to the street class with the ArrayList<E>.



```java
public boolean addBuilding(ArrayList<Building> row, Building newBuilding)
{
    checkAddBuildingValidity(row, newBuilding);
    row.add(newBuilding);
    return true;
}
```

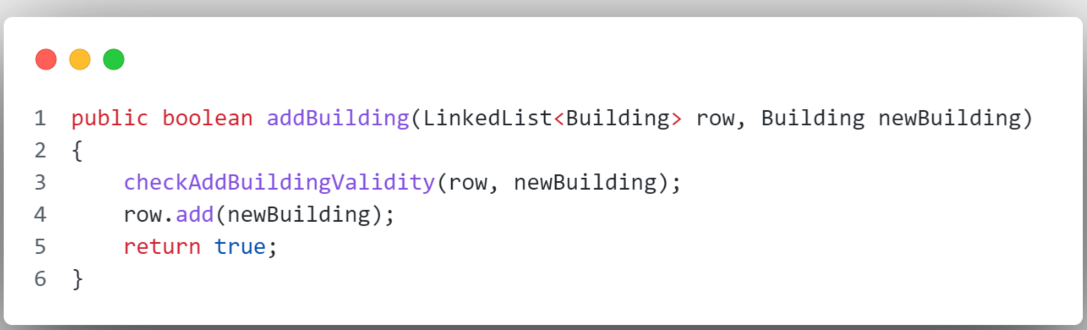
checkAddBuildingValidity() will be *O(n)*.

If there is enough space in the row, the building will be add in constant time to the end of the list. *θ(1)*
Otherwise, add() method of ArrayList will increase its size twice and replace previous items to the new list and adds the Building to the list. *θ(n)*

It is normally linear time, but this will not be happen always. (Mostly there will be space at the tail)
Therefor, the total time complexity of insertion operation will be, *amortized θ(1)*

So, since the check is *O(n)* the time complexity of addBuilding() method for this Street class will be: O(n)

Add a building to the street class with the LinkedList<E>.

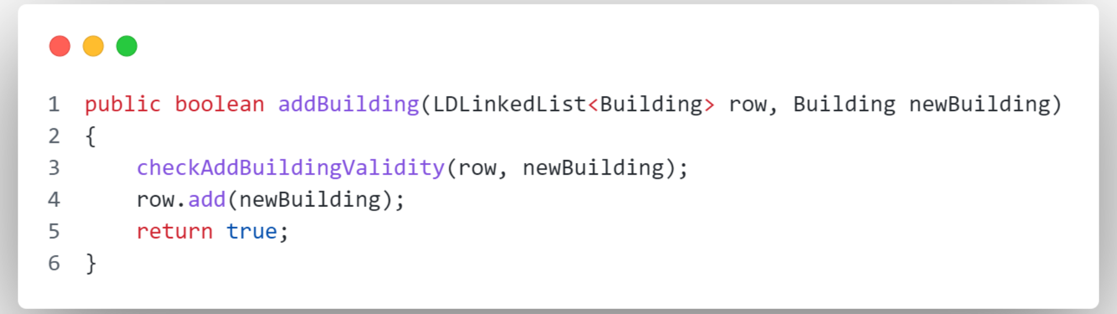


```java
public boolean addBuilding(LinkedList<Building> row, Building newBuilding)
{
    checkAddBuildingValidity(row, newBuilding);
    row.add(newBuilding);
    return true;
}
```

checkAddBuildingValidity() will be *O(n)*.
add() method of LinkedList is constant. *θ(1)*

So, since the check is *O(n)* the time complexity of addBuilding() method for this Street class will be: O(n)

Add a building to the street class with the LinkedList<E>.



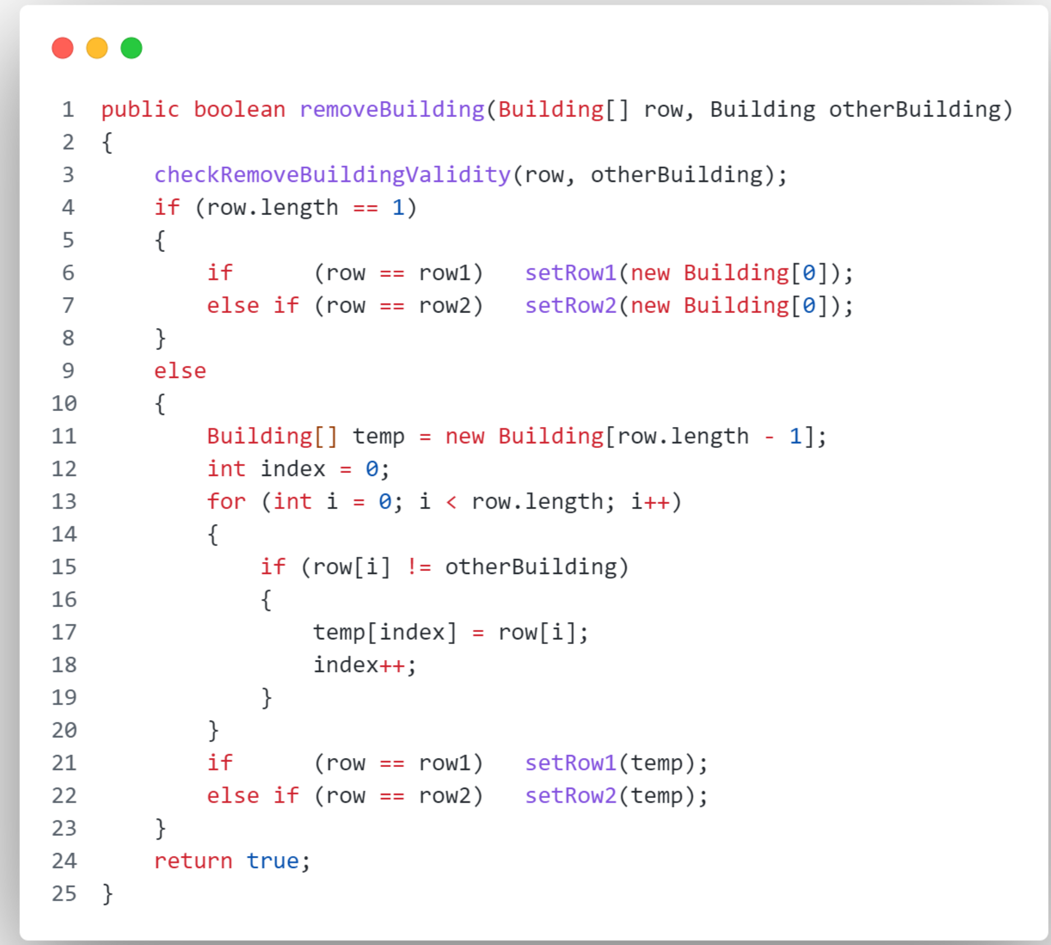
```java
public boolean addBuilding(LDLinkedList<Building> row, Building newBuilding)
{
    checkAddBuildingValidity(row, newBuilding);
    row.add(newBuilding);
    return true;
}
```

checkAddBuildingValidity() will be *O(n)*.
add() method of LDLinkedList is constant. *θ(1)*

[LDLinkledList implementation will be shown later of this document]
So, since the check is *O(n)* the time complexity of addBuilding method for this Street class will be: O(n)

- Impelementations of removeBuildings() methods.

<u>Remove a building from the street class with the basic array.</u>



```java
public boolean removeBuilding(Building[] row, Building otherBuilding)
{
    checkRemoveBuildingValidity(row, otherBuilding);
    if (row.length == 1)
    {
        if      (row == row1)   setRow1(new Building[0]);
        else if (row == row2)   setRow2(new Building[0]);
    }
    else
    {
        Building[] temp = new Building[row.length - 1];
        int index = 0;
        for (int i = 0; i < row.length; i++)
        {
            if (row[i] != otherBuilding)
            {
                temp[index] = row[i];
                index++;
            }
        }
        if      (row == row1)   setRow1(temp);
        else if (row == row2)   setRow2(temp);
    }
    return true;
}
```

To remove a Building from the Street, we should check first remove can be performed.

method checkRemoveBuildingValidity (), checks street length *θ(1)*,
checks if there is a Building in the row, and if building exist in the row.

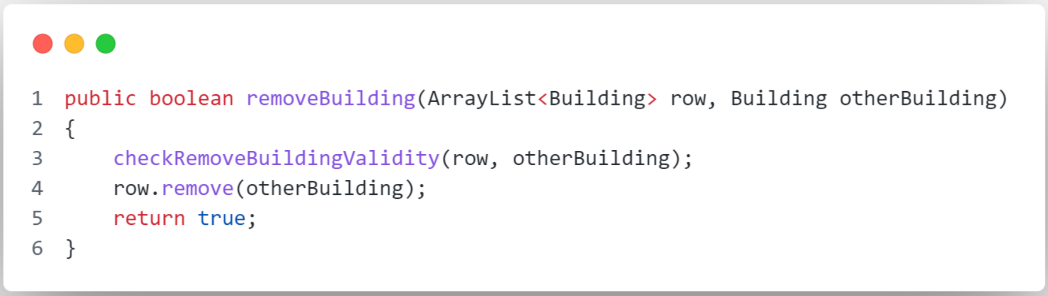
Controling existancy will be depend on number of Buildings on the row: *θ(n)*

If all conditions satisfies and building removable, a new array will be created with one small size and remaining elements will be copies into this new array: *θ(n)*

So, the total complexity of this removeBuildings() for basic array will be: O(n)

Meanwhile, time of checkRemoveBuildingValidity() will be O(n) for all containers.

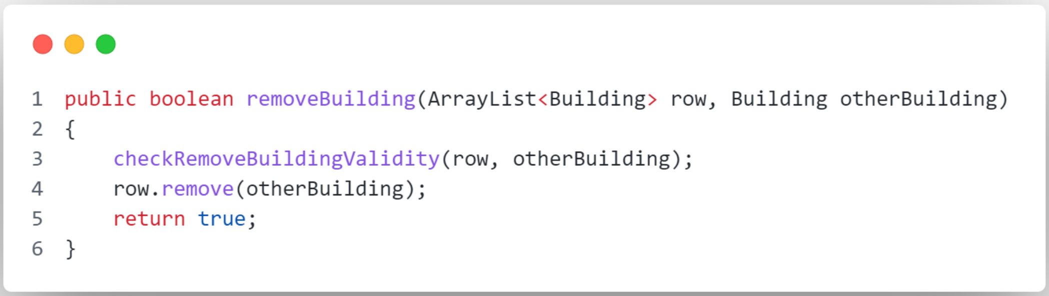
Remove a building from the street class with the ArrayList<E>.



```
public boolean removeBuilding(ArrayList<Building> row, Building otherBuilding)
{
    checkRemoveBuildingValidity(row, otherBuilding);
    row.remove(otherBuilding);
    return true;
}
```

checkRemoveBuildingValidity() method will take linear time: *O(n)*
Remove operation of ArrayList<E> is linear. *θ(n)*
Therefor, the opration will be O(n).

Remove a building from the street class with the LinkedList<E>.



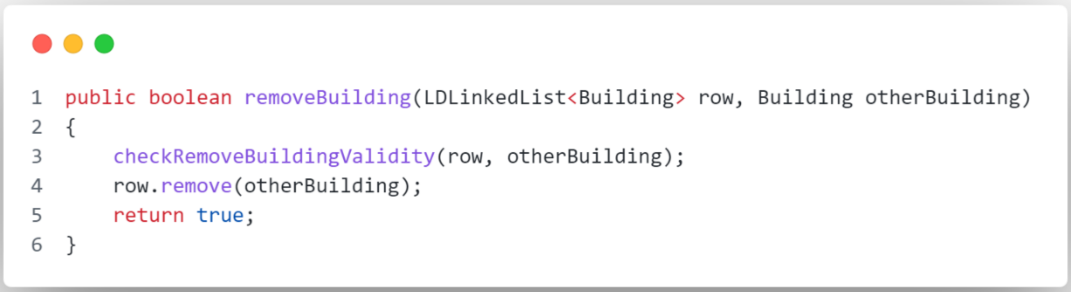
```
public boolean removeBuilding(ArrayList<Building> row, Building otherBuilding)
{
    checkRemoveBuildingValidity(row, otherBuilding);
    row.remove(otherBuilding);
    return true;
}
```

checkRemoveBuildingValidity() method will be linear: *O(n)*

Remove operation of LinkedList<E> is linear, because it takes the element and search it. *θ(n)*

Therefor, the opration will be O(n).

Remove a building from the street class with the LDLinkedList<E>.



```java
public boolean removeBuilding(LDLinkedList<Building> row, Building otherBuilding)
{
    checkRemoveBuildingValidity(row, otherBuilding);
    row.remove(otherBuilding);
    return true;
}
```

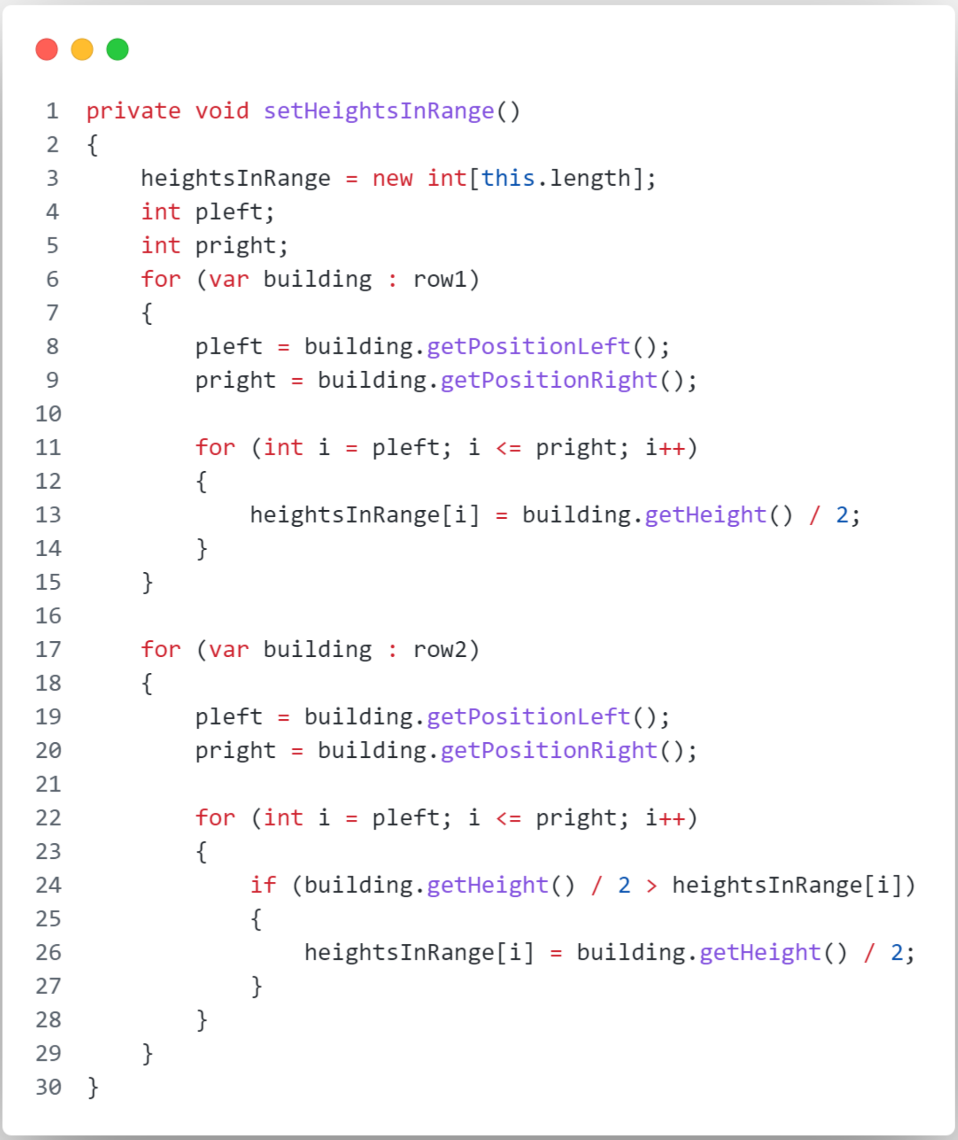
checkRemoveBuildingValidity() method will be linear: *O(n)*
Remove operation of LinkedListLD<E> is linear, because it takes the element and search it. *θ(n)*

Therefor, the opration will be O(n).

- Impelementations of setHeightsInRange() methods.

  It is same for all Street classes

  <u>Set heights in range of street class</u>



```java
private void setHeightsInRange()
{
    heightsInRange = new int[this.length];
    int pleft;
    int pright;
    for (var building : row1)
    {
        pleft = building.getPositionLeft();
        pright = building.getPositionRight();

        for (int i = pleft; i <= pright; i++)
        {
            heightsInRange[i] = building.getHeight() / 2;
        }
    }

    for (var building : row2)
    {
        pleft = building.getPositionLeft();
        pright = building.getPositionRight();

        for (int i = pleft; i <= pright; i++)
        {
            if (building.getHeight() / 2 > heightsInRange[i])
            {
                heightsInRange[i] = building.getHeight() / 2;
            }
        }
    }
}
```

setHeightsInRange() method iterates twice as much as the number of Buildings in first and second row, respectively.
It will fill an array with the heighest values in the street at current position.
So, iteration will take as much as the Building numbers in the rows, *θ(n)*

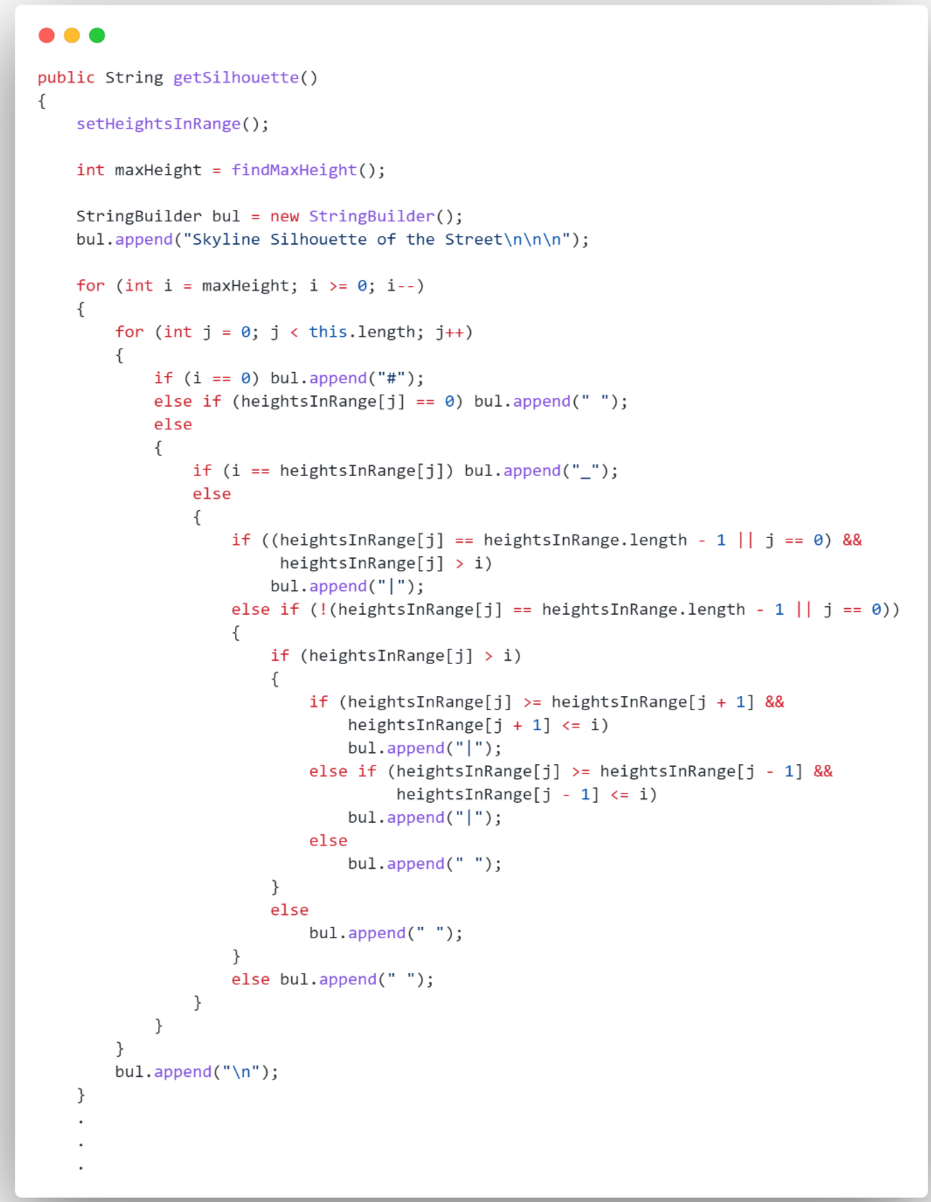
and also filling array will take as much as the length of the building. *θ(n)*

Length of the building will averagly be same for the buldings, therefor, this methods complexity is *amortized linear time*. O(n)

- Impelementations of getSilhouette() methods.

  It is same for all Street classes.

  Get silhouette of the street class

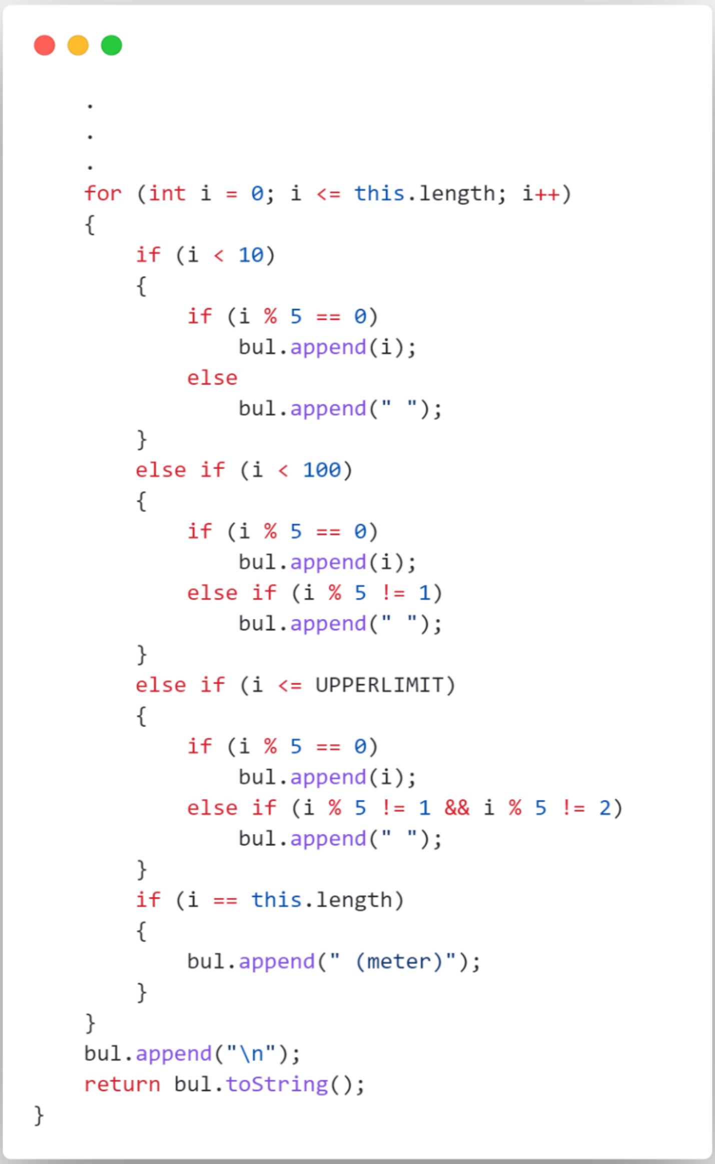


```
public String getSilhouette()
{
    setHeightsInRange();

    int maxHeight = findMaxHeight();

    StringBuilder bul = new StringBuilder();
    bul.append("Skyline Silhouette of the Street\n\n\n");

    for (int i = maxHeight; i >= 0; i--)
    {
        for (int j = 0; j < this.length; j++)
        {
            if (i == 0) bul.append("#");
            else if (heightsInRange[j] == 0) bul.append(" ");
            else
            {
                if (i == heightsInRange[j]) bul.append("_");
                else
                {
                    if ((heightsInRange[j] == heightsInRange.length - 1 || j == 0) &&
                         heightsInRange[j] > i)
                        bul.append("|");
                    else if (!(heightsInRange[j] == heightsInRange.length - 1 || j == 0))
                    {
                        if (heightsInRange[j] > i)
                        {
                            if (heightsInRange[j] >= heightsInRange[j + 1] &&
                                heightsInRange[j + 1] <= i)
                                bul.append("|");
                            else if (heightsInRange[j] >= heightsInRange[j - 1] &&
                                     heightsInRange[j - 1] <= i)
                                bul.append("|");
                            else
                                bul.append(" ");
                        }
                        else
                            bul.append(" ");
                    }
                    else bul.append(" ");
                }
            }
        }
        bul.append("\n");
    }
    .
    .
    .
```



```
    .
    .
    .
    for (int i = 0; i <= this.length; i++)
    {
        if (i < 10)
        {
            if (i % 5 == 0)
                bul.append(i);
            else
                bul.append(" ");
        }
        else if (i < 100)
        {
            if (i % 5 == 0)
                bul.append(i);
            else if (i % 5 != 1)
                bul.append(" ");
        }
        else if (i <= UPPERLIMIT)
        {
            if (i % 5 == 0)
                bul.append(i);
            else if (i % 5 != 1 && i % 5 != 2)
                bul.append(" ");
        }
        if (i == this.length)
        {
            bul.append(" (meter)");
        }
    }
    bul.append("\n");
    return bul.toString();
}
```

Before the creating silhouette, highest points will be marked by setHeightsInRange() It's *amortized* linear time: *θ(n)*

In getSilhouette() StringBuilder object is used. It keep reference of every item that would be print to the screen. It works in linear time: *θ(n)*

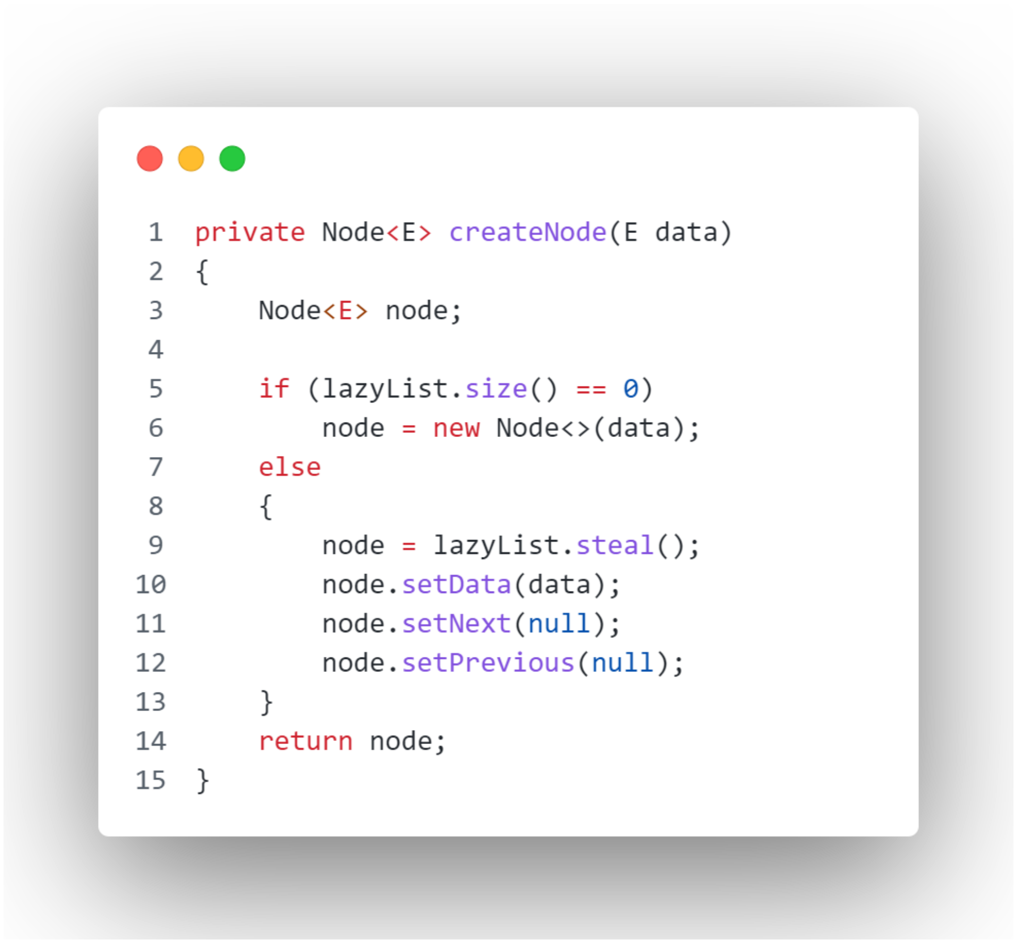
There are 2 nested iteration, one of them iterates as much as highest point of street *θ(h)*, other iterates as much as length of street *θ(l)*. Other operation inside the loops are constant. (comparision,

assigment etc.) Therefor, time complexity of loops will be *θ(h*l)*
BURAYA SONUCLARDAN SONRA EKLEME YAP!!!

- Impelementations of LDLinkedList class

  Here, we will look at the basic LDLinkedList methods

  createNode(E data) method

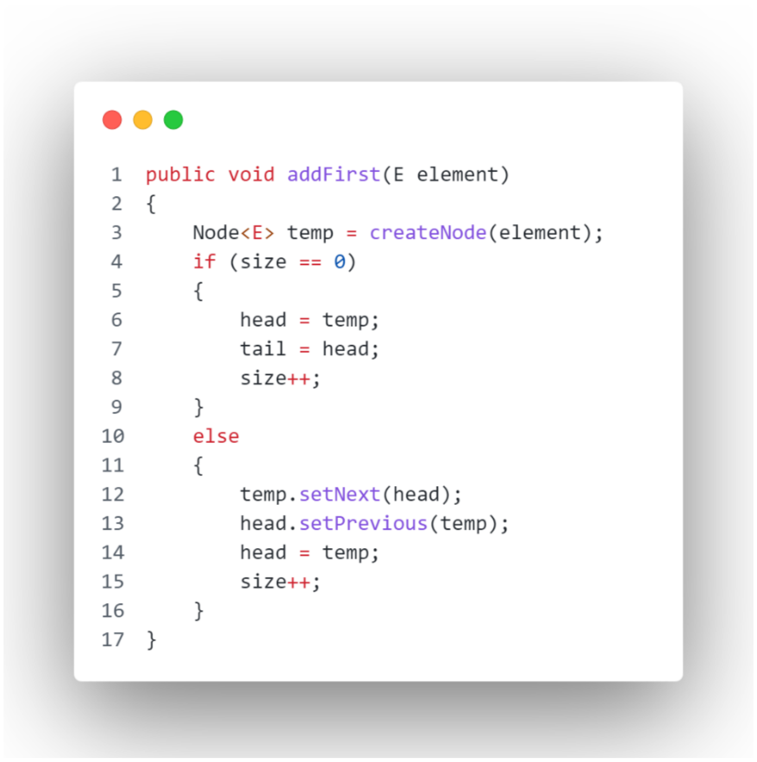


```java
private Node<E> createNode(E data)
{
    Node<E> node;

    if (lazyList.size() == 0)
        node = new Node<>(data);
    else
    {
        node = lazyList.steal();
        node.setData(data);
        node.setNext(null);
        node.setPrevious(null);
    }
    return node;
}
```

If the lazyList is not empty, it takes first element of lazyList and create node by stealing its memory, otherwise, creates a new node with the data.
It is: θ(1)

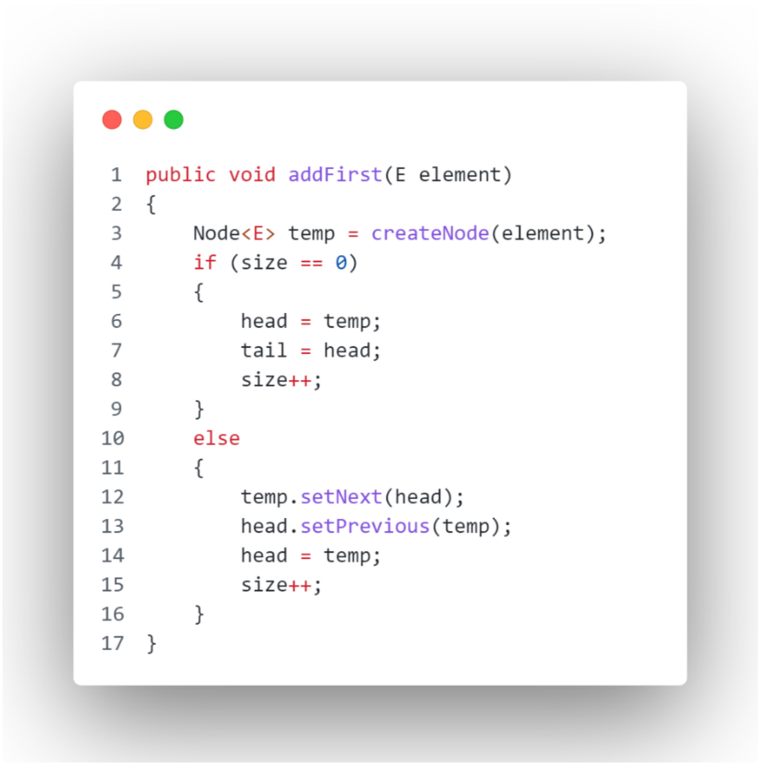
addFirst(E element) method



```java
public void addFirst(E element)
{
    Node<E> temp = createNode(element);
    if (size == 0)
    {
        head = temp;
        tail = head;
        size++;
    }
    else
    {
        temp.setNext(head);
        head.setPrevious(temp);
        head = temp;
        size++;
    }
}
```

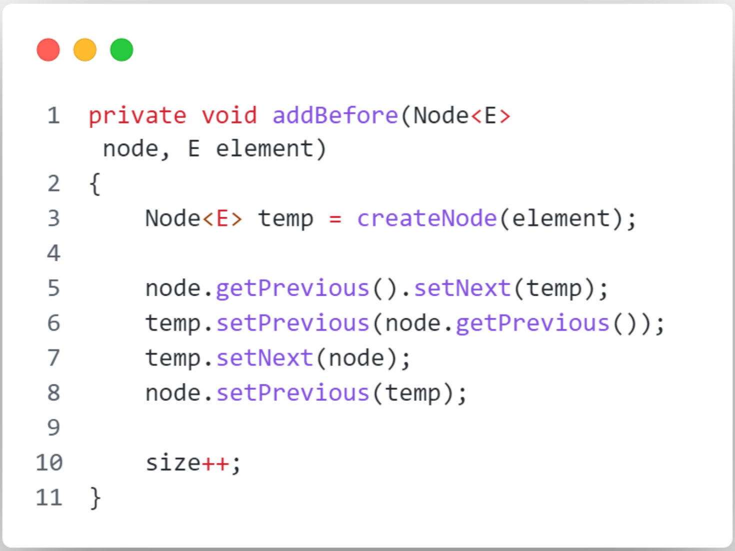
Creates node *θ(1)*, and adds it to the head of the list. θ(1)

addLast(E element) method



```java
public void addFirst(E element)
{
    Node<E> temp = createNode(element);
    if (size == 0)
    {
        head = temp;
        tail = head;
        size++;
    }
    else
    {
        temp.setNext(head);
        head.setPrevious(temp);
        head = temp;
        size++;
    }
}
```

Creates node *θ(1)*, and adds it to the head of the list. θ(1), addBefore(Node<E>, E element) method



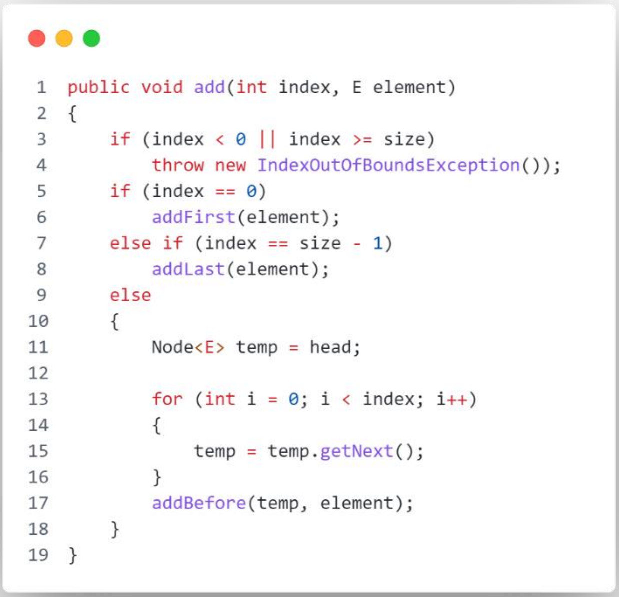
```java
private void addBefore(Node<E>
 node, E element)
{
    Node<E> temp = createNode(element);

    node.getPrevious().setNext(temp);
    temp.setPrevious(node.getPrevious());
    temp.setNext(node);
    node.setPrevious(temp);

    size++;
}
```

Creates node *θ(1),* and adds it to the before node that took as parameter. θ(1)

add(int index, E element) method

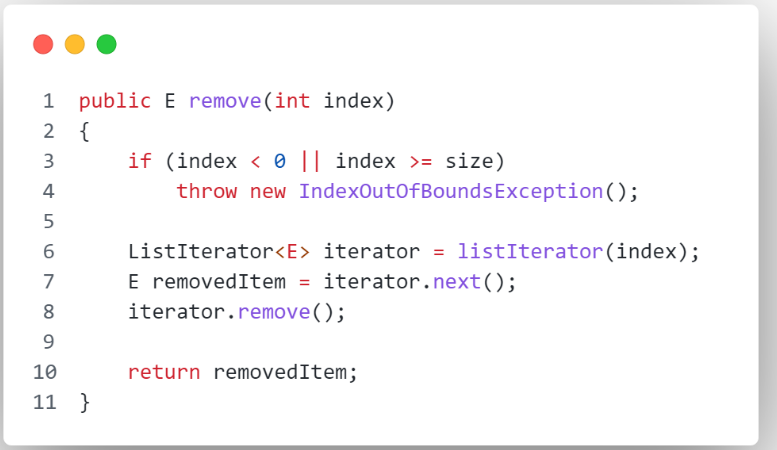


```java
public void add(int index, E element)
{
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException());
    if (index == 0)
        addFirst(element);
    else if (index == size - 1)
        addLast(element);
    else
    {
        Node<E> temp = head;

        for (int i = 0; i < index; i++)
        {
            temp = temp.getNext();
        }
        addBefore(temp, element);
    }
}
```

Takes index and invoke methods -*θ(1)*- to add appropriate positions, if index not head or tail it will be linear. *O(n)*

remove(int index)



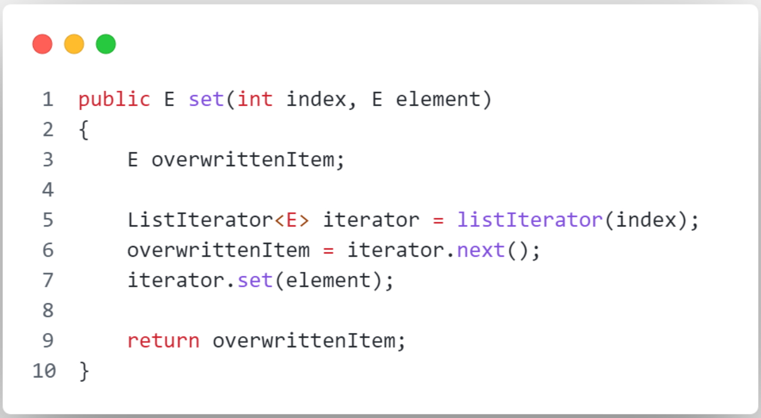
```java
public E remove(int index)
{
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();

    ListIterator<E> iterator = listIterator(index);
    E removedItem = iterator.next();
    iterator.remove();

    return removedItem;
}
```

It iterates as much as the index, finds node and remove it by iterator method
<Iterator implementations will be shown below>
So search will make it linear. O(n)

set(int index, E element)



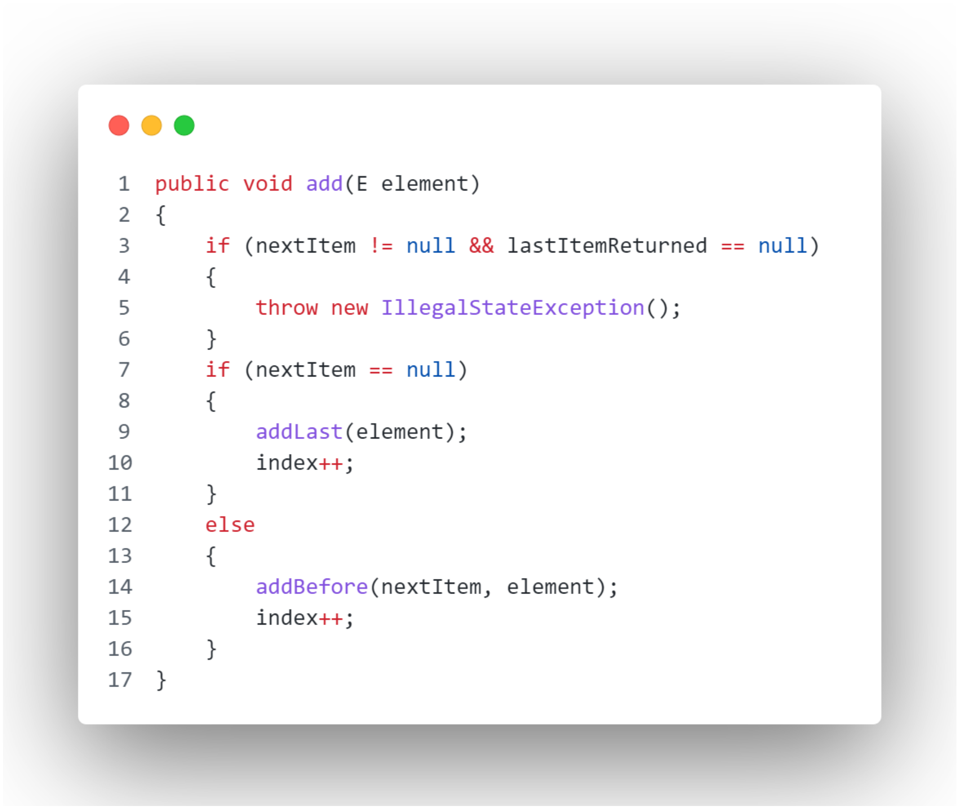
```java
public E set(int index, E element)
{
    E overwrittenItem;

    ListIterator<E> iterator = listIterator(index);
    overwrittenItem = iterator.next();
    iterator.set(element);

    return overwrittenItem;
}
```

It will take linear time to find the index *θ(n)* .
Set function of iterator is constant. *θ(1)*
<Iterator implementations will be shown below>
Total complexity will be θ(1)

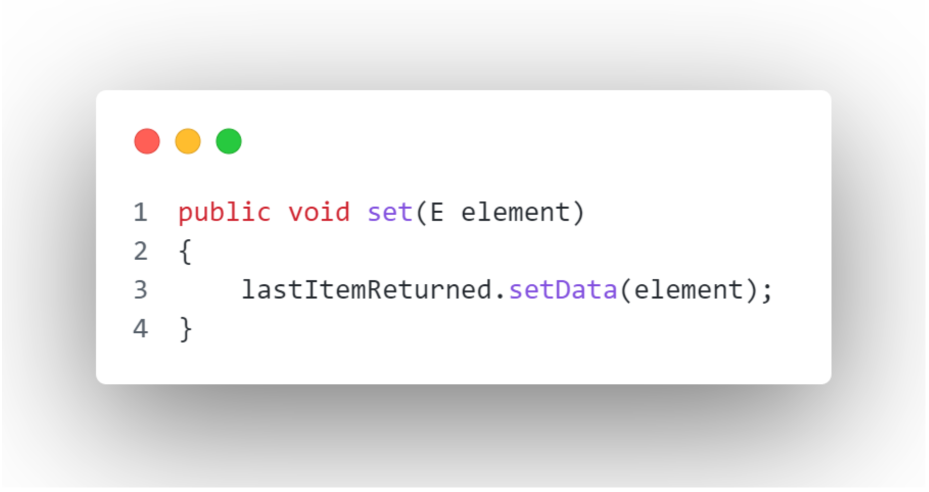
ListIterator add(E element) method



```java
public void add(E element)
{
    if (nextItem != null && lastItemReturned == null)
    {
        throw new IllegalStateException();
    }
    if (nextItem == null)
    {
        addLast(element);
        index++;
    }
    else
    {
        addBefore(nextItem, element);
        index++;
    }
}
```

Adds element just before the last item that returned by next().
It is constant. θ(1)

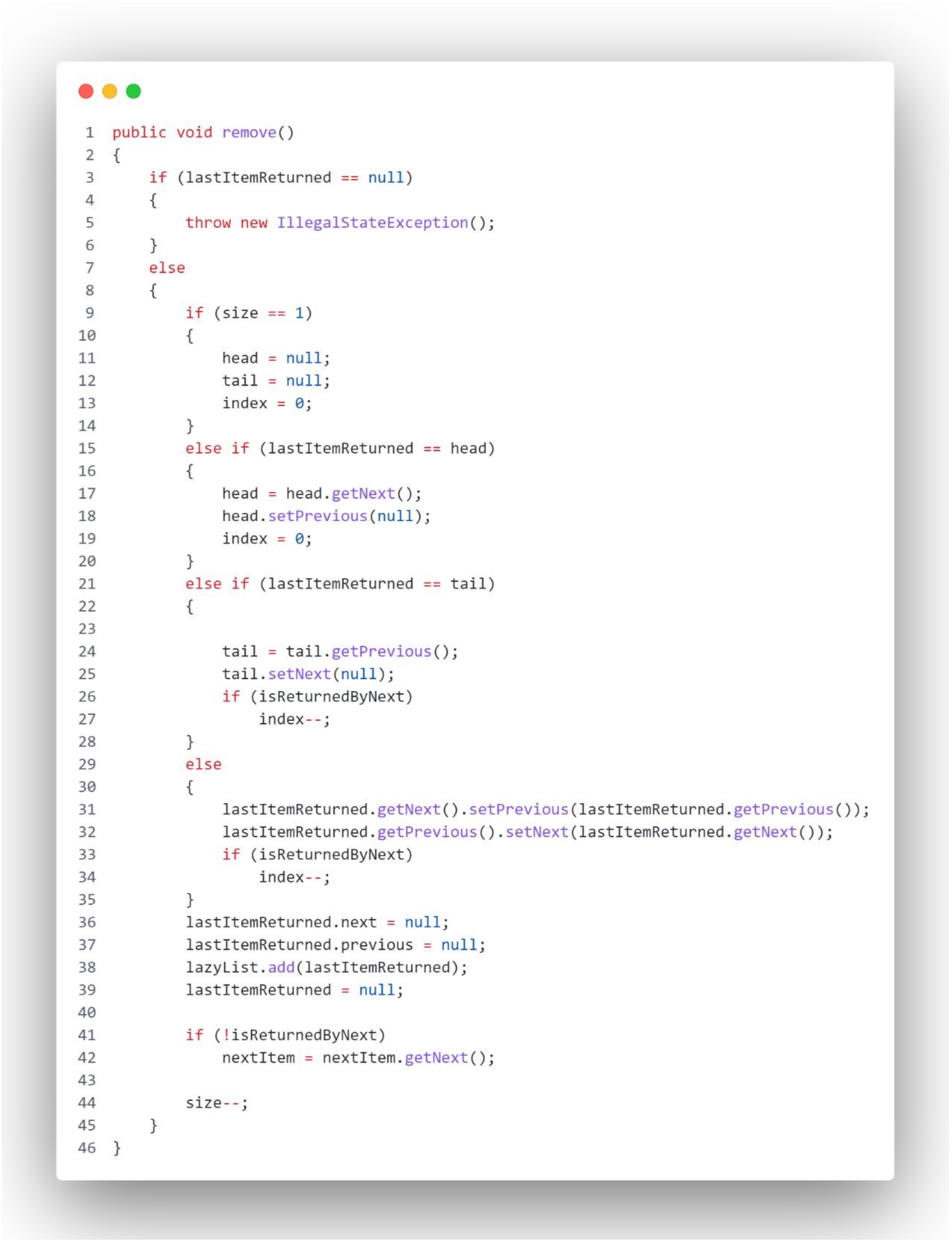
ListIterator set(E element) method



```java
public void set(E element)
{
    lastItemReturned.setData(element);
}
```

Change data of last item that returned by next().
It is constant. θ(1)

<u>ListIterator remove() method</u>



```java
public void remove()
{
    if (lastItemReturned == null)
    {
        throw new IllegalStateException();
    }
    else
    {
        if (size == 1)
        {
            head = null;
            tail = null;
            index = 0;
        }
        else if (lastItemReturned == head)
        {
            head = head.getNext();
            head.setPrevious(null);
            index = 0;
        }
        else if (lastItemReturned == tail)
        {

            tail = tail.getPrevious();
            tail.setNext(null);
            if (isReturnedByNext)
                index--;
        }
        else
        {
            lastItemReturned.getNext().setPrevious(lastItemReturned.getPrevious());
            lastItemReturned.getPrevious().setNext(lastItemReturned.getNext());
            if (isReturnedByNext)
                index--;
        }
        lastItemReturned.next = null;
        lastItemReturned.previous = null;
        lazyList.add(lastItemReturned);
        lastItemReturned = null;

        if (!isReturnedByNext)
            nextItem = nextItem.getNext();

        size--;
    }
}
```

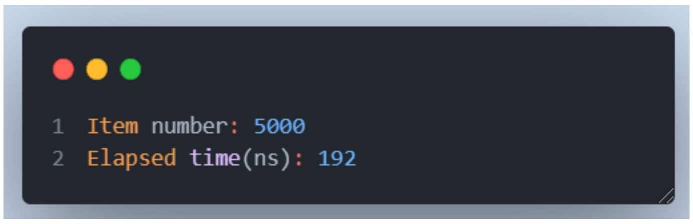
It removes next element that will be return if next() will be invoke, or previous element that will be return if previous() will be invoke. *θ(1)*
Also, add removed element to the lazy list, which is a linked list and addition is constant. *θ(1)*
Total complexity is constant. θ(1)

4. Test Cases

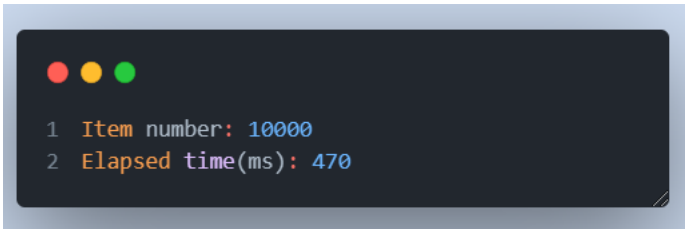
Add and remove buildings to each street classes, 5000, 10000, 20000, 40000 times, and compare complexity.

5. Running Command and Results

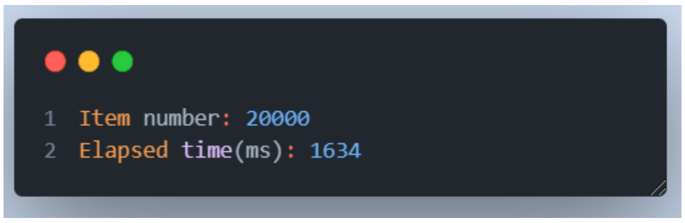
Result of adding building to the basic array:



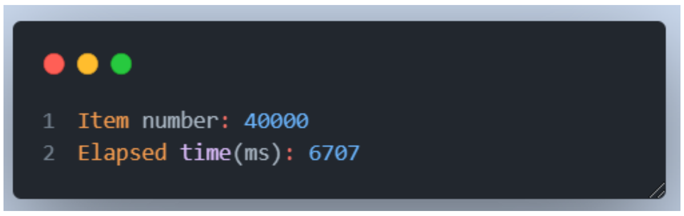
```
Item number: 5000
Elapsed time(ns): 192
```



```
Item number: 10000
Elapsed time(ms): 470
```

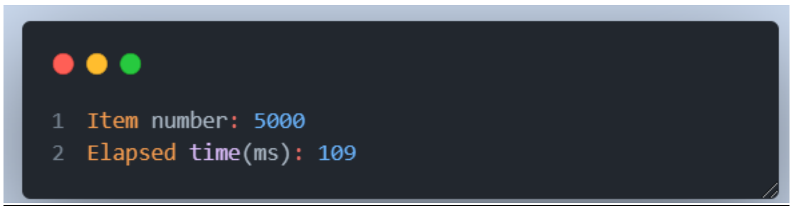


```
Item number: 20000
Elapsed time(ms): 1634
```

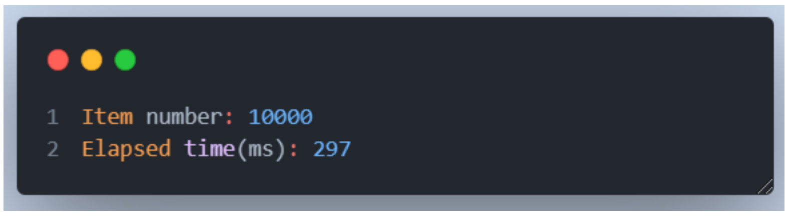


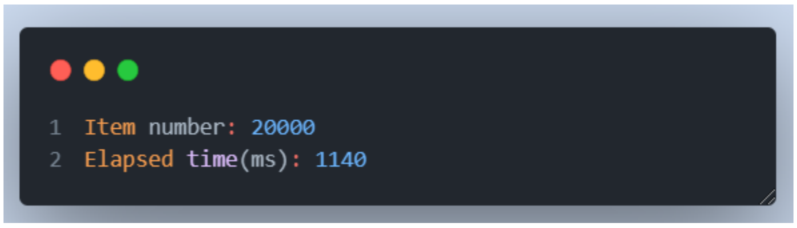
```
Item number: 40000
Elapsed time(ms): 6707
```

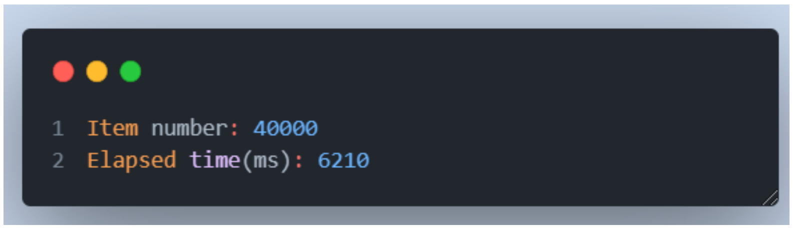
Result of adding building to the ArrayList:



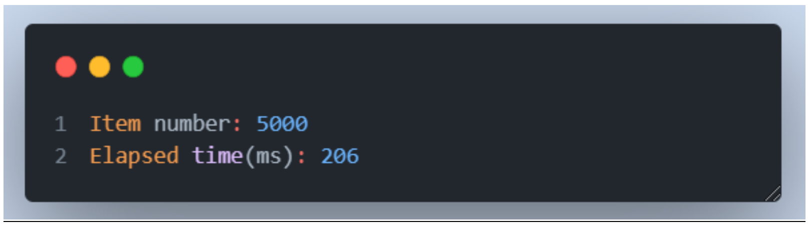
```
Item number: 5000
Elapsed time(ms): 109
```

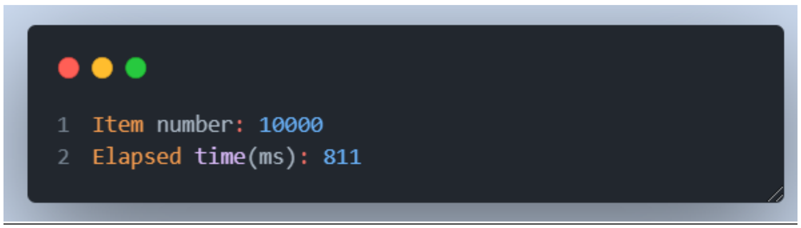

```
Item number: 10000
Elapsed time(ms): 297
```

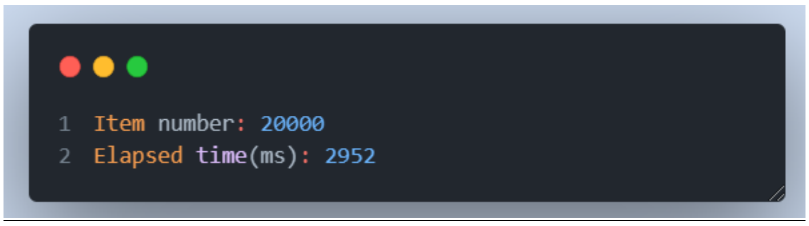

```
Item number: 20000
Elapsed time(ms): 1140
```

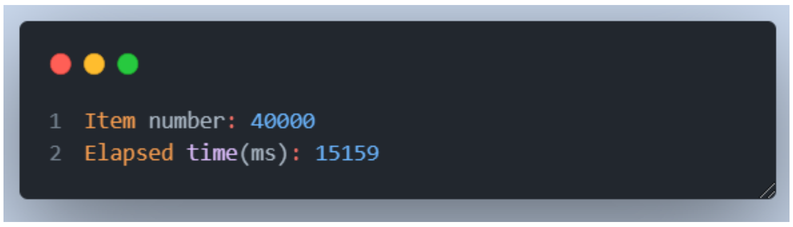

```
Item number: 40000
Elapsed time(ms): 6210
```

<u>Result of adding building to the LinkedList:</u>


```
Item number: 5000
Elapsed time(ms): 206
```

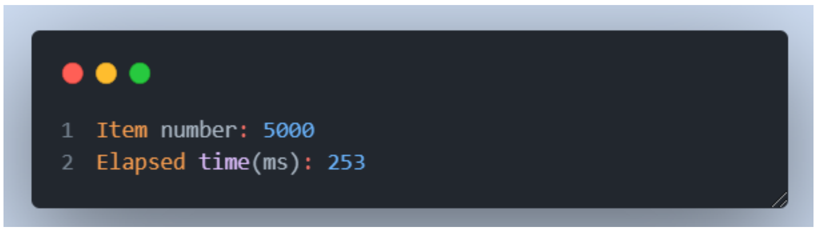

```
Item number: 10000
Elapsed time(ms): 811
```


```
Item number: 20000
Elapsed time(ms): 2952
```

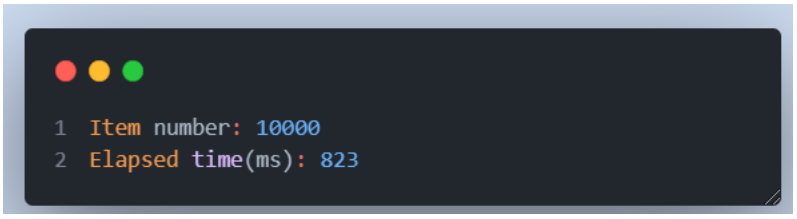


```
Item number: 40000
Elapsed time(ms): 15159
```

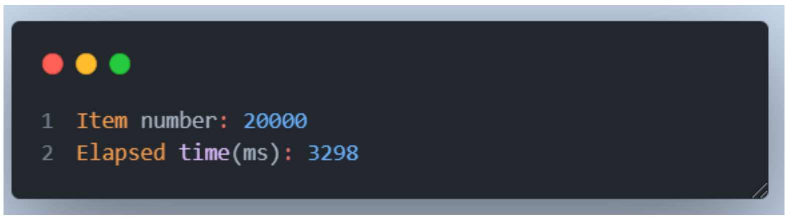
Result of adding building to the LDLinkedList:



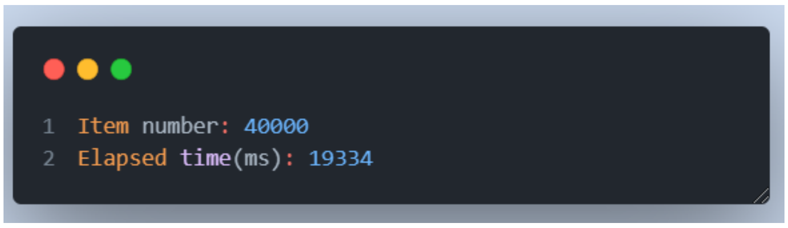
```
Item number: 5000
Elapsed time(ms): 253
```



```
Item number: 10000
Elapsed time(ms): 823
```

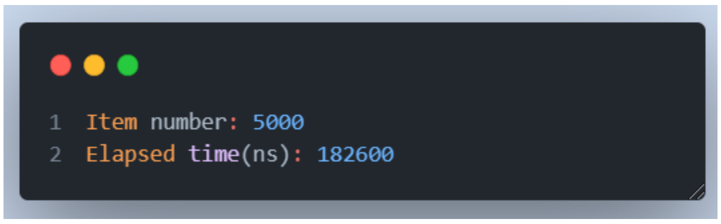


```
Item number: 20000
Elapsed time(ms): 3298
```

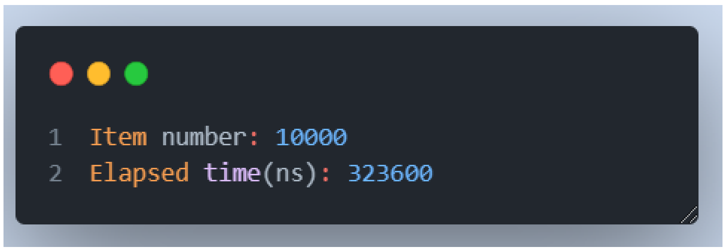


```
Item number: 40000
Elapsed time(ms): 19334
```

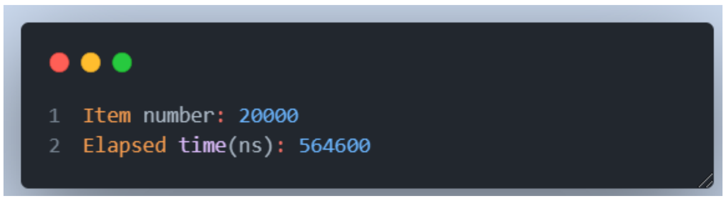
Result of removing building to the basic array:



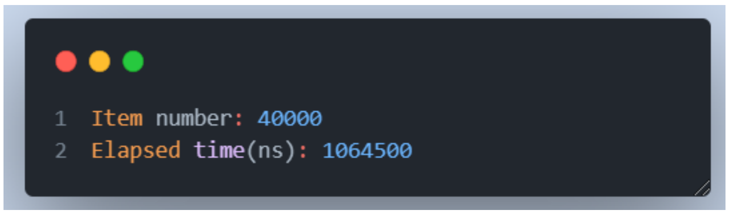
```
Item number: 5000
Elapsed time(ns): 182600
```



```
Item number: 10000
Elapsed time(ns): 323600
```

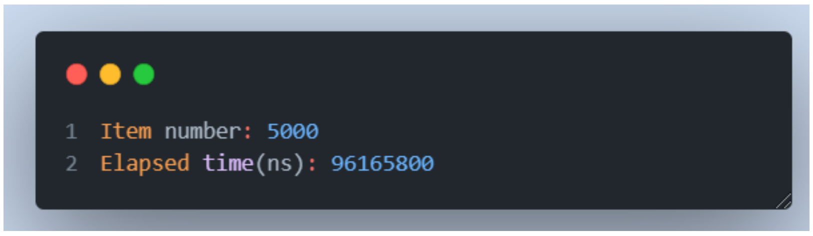


```
Item number: 20000
Elapsed time(ns): 564600
```

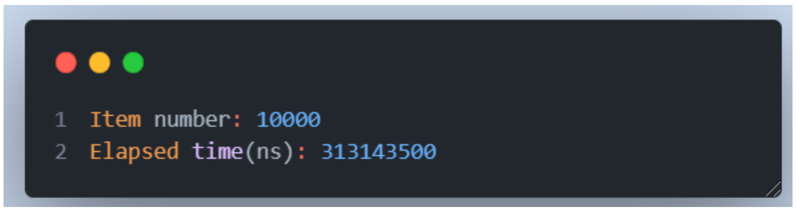


```
Item number: 40000
Elapsed time(ns): 1064500
```

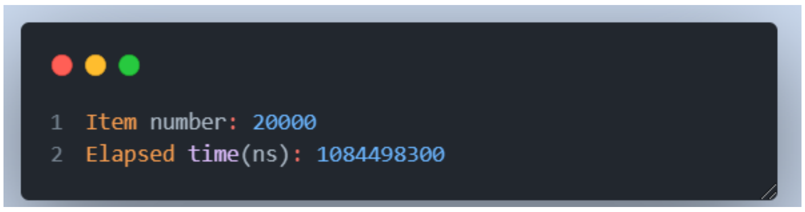
Result of removing building to the ArrayList:



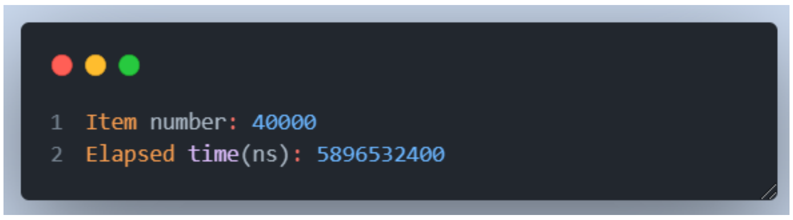
```
Item number: 5000
Elapsed time(ns): 96165800
```



```
Item number: 10000
Elapsed time(ns): 313143500
```

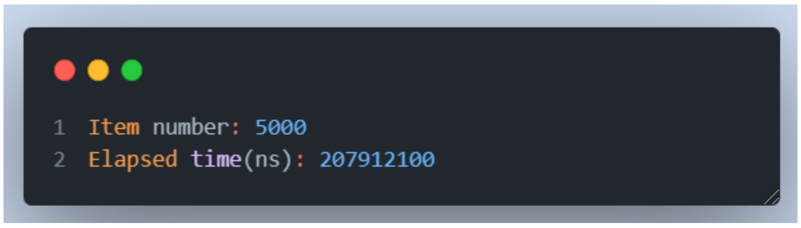


```
Item number: 20000
Elapsed time(ns): 1084498300
```

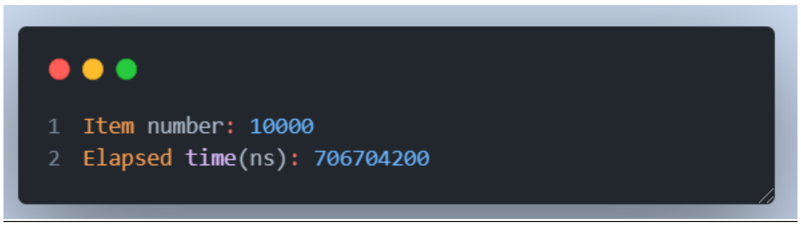


```
Item number: 40000
Elapsed time(ns): 5896532400
```

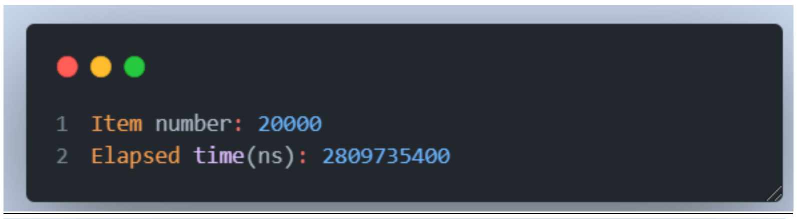
Result of removing building to the LinkedList:



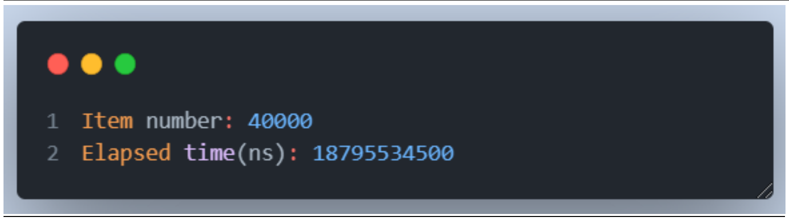
```
Item number: 5000
Elapsed time(ns): 207912100
```



```
Item number: 10000
Elapsed time(ns): 706704200
```



```
Item number: 20000
Elapsed time(ns): 2809735400
```



```
Item number: 40000
Elapsed time(ns): 18795534500
```